MASSACHUSETTS INSTITUTE OF TECHNOLOGY

6.096: Introduction to C++ IAP 2009

PROBLEM SET 3

Function prototypes:
An additional point about function prototypes that we did not emphasize in lecture is that you do not need a separate function prototype if you define the entire function above the point where it is called. If you define `square` before `main`, for instance, you do not need a prototype for `square`.

A note on global variables:
We discussed in lecture how variables can be declared at *file scope* – if a variable is declared outside of any function, it can be used anywhere in the file. For anything besides global constants, this is usually a bad idea – if you need to access the same variable from multiple functions, most often you simply want to pass the variable around as an argument between the functions. Avoid global variables when you can.

Part 1 – Fix the Function:

1. Identify the errors in the following programs, and explain how you would correct them to make them do what they were apparently meant to do.

   a.
```
#include <iostream>

int main() {
      std::cout << square(35);
      return 0;
}

int square(int number) { return number * number; }
```

   b.
```
#include <iostream>

int square();

int main() {
      int number = 35;
      std::cout << square(number);
      return 0;
}

int square() { return number * number; }
```

   (Give two ways to make this program work. Indicate which is preferable and why.)

c.
```
#include <iostream>

void square(int number);

int main() {
      int numberToSquare = 35;
      square(numberToSquare);
      std::cout << numberToSquare;  // Should print 35^2
      return 0;
}

void square(int number) { number = number * number; }
```

(For this problem, do not change the return type of the `square` function.)

d.
```
#include <iostream>

int sumOfPositives(int a, int b);

int main() {
      std::cout << sumOfPositives(35, 25, 3);
      return 0;
}

int sumOfPositives(int a, int b) {
      int sum = 0;
      if(a > 0)
            sum += a;
      if(b > 0)
            sum += b;
}
```

(Two mistakes, one of which you may fix in either of two ways.)

2. What do the following program fragments do? (Try to answer without compiling and running them.) Briefly explain your results.

   a.
```
char *hello = "Hello world!";

void printHello(char *hello) {
      std::cout << hello;
}

int main() {
      ...
      printHello("This is not the hello message!");
      ...
}
```

b. `int mystery(int &number) { std::cout << number; return ++number; }`

```
int main() {
      for(int i = 0; i < 10; i++) {
            cout << mystery(i);
      }
      ...
}
```

c. `int mystery(int number) { std::cout << number; return ++number; }`

```
int main() {
      for(int i = 0; i < 10; i++) {
            cout << mystery(i);
      }
      ...
}
```

Part 2 – Random numbers:

Random number generation:
C++ provides the `rand()` function, which returns a pseudorandom integer between 0 and `RAND_MAX` (a constant integer defined by the compiler). The integer returned is not truly random because the random number generator actually uses an initial "seed" value to determine a fixed sequence of almost random numbers; the sequence is the same each time the same seed is used.

To initialize the random number generator, we call the `srand` function with the desired seed as an argument. This only needs to be called once in the entire program (having one-time-per-program initializer functions is not uncommon). The seed passed to the function is often the current time, which changes on each execution and is obtained via the `time(0)` function call. Once the initialization has been performed, we can just call `rand()` each time we want another random number.

Putting it all together, generating a random number looks something like this:

```
#include <iostream>
#include <cstdlib>   //C standard library - defines rand(), srand(), RAND_MAX
#include <ctime>     //C time functions - defines time()

int main() {
      srand( time(0) ); // Initialize random number generator
      int random1 = rand(), random2 = rand();
      cout << "random1: " << random1 << "\nrandom2: " << random2 << endl;
}
```

If we want to restrict the random numbers to a particular range, we can use the modulus (%) operator as follows:

```
int dieRollValue  = rand() % 6 + 1;
```

The `%6` ensures that the number is between 0 and 5, and adding one shifts this to be between 1 and 6 (the appropriate range for a die roll). (Recall that `a % b` returns the remainder of `a` / `b`.)

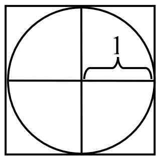
We can also convert the random integers to decimal values in the range [0, 1] simply by dividing by `RAND_MAX`:

```
double randomDecimal = rand() / static_cast<double>(RAND_MAX);
```

The `static_cast` keyword creates a temporary copy of the variable in parentheses, where the copy is of the type indicated in angle brackets. This ensures that our division is done as floating-point, and not integer division. This is called "casting" `RAND_MAX` to a double.

Potshots at Pi:
Let's calculate pi! We're not going to do it by measuring circumferences, though; we're going to do it with Monte Carlo methods – using random numbers to run a simulation.

Let's take a circle of radius 1 around the origin, circumscribed by a square with side length 2, like so:



Imagine that this square is a dartboard, and that you are tossing many, many darts at it at random locations on the board. With enough darts, the ratio of darts in the circle to total darts thrown should be the ratio between the area of the circle and the area of the square. Since you know the area of the square, you can use this ratio to calculate the area of the circle, from which you can calculate pi using $\pi = \frac{a}{r^2}$.

1. We can make the math a little easier. All we really need to calculate is the area of one quadrant and multiply by 4 to get the full area. This allows us to restrict the domain of our x and y coordinates to [0,1].
    a. Write two variable declarations representing the x-coordinate and y-coordinate of a particular dart throw. Each one should be named appropriately, and should be initialized to a random `double` in the range [0,1].
    b. Place your variable declarations within a loop that increments a variable named `totalDartsInCircle` if the dart's location is within the circle's radius. You'll need to use the Euclidean distance formula ($d^2 = x^2 + y^2$). (Assume for the moment that `totalDartsInCircle` has already been declared.)
    c. Now use your loop to build a function that takes one argument, specifying the number of "dart throws" to run, and returns the decimal value of pi, using the technique outlined above. Be sure to name your function appropriately. You should get pretty good results for around 5,000,000 dart throws.

Function overloading:
C++ supports a very powerful feature called *function overloading*. This means that you can define two functions by the same name, but with different argument lists. The argument lists can differ by number of

arguments, argument type, or both. When you call the function, the compiler will examine the arguments that you pass and automatically select the function whose prototype matches the arguments specified. If there is no function whose prototype matches the call, or there is more than one (i.e. the call is ambiguous), the compiler will spit out a syntax error.

2. Wouldn't it be nice if we didn't have to do all this modulus and subtraction stuff each time we wanted a random number? Write two functions, both called `randomInRange`, to simplify random number generation. One function should take an upper and lower bound on the range of allowed integers and return a random integer in that range. The other should do the same thing for `double`s. Test your functions to make sure they produce correct results.
   Note: The actual random number range for both functions should be [lower, upper]. Your code will be slightly different for the `int` *and* `double` *versions.*

3. Why, given your definitions for `randomInRange`, is `randomInRange(1, 10.0)` a syntax error?
   Hint: Think about promotion and demotion – the conversion of arguments between types in a function call.

<u>Part 3 – Maxing out:</u>

1.
   a. Write a single `maximum` function that returns the maximum of two integers. (The ternary operator, though not necessary, is a very elegant way to do this in one line.)
   b. Write a set of 3 functions capable, collectively, of finding the maximum of anywhere between 2 and 4 numbers. Each one should take integer arguments and return an integer. The function call required to find any such maximum should look something like `maximum(40, 24, 83)`.
      Hint: Use overloading, and think about how you can build larger functions out of smaller ones.
2. Write a single maximum function capable of handling an arbitrary number of integers to compare. It should take two arguments, and return an integer.
   Hint: What data types can you use to represent an arbitrarily large set of integers in two arguments?
3. Use default arguments to define a single `maximum` function that can be called with anywhere from 2 to 4 explicit arguments. In defining your default values, you may want to take advantage of the constant `INT_MIN`, defined in standard include file `climits`, which is a compiler-specific constant holding the lowest possible integer value.

<u>Templates:</u>
Function overloading allows us to define *different but similar* operations to perform on different argument types. Sometimes, though, you may find that you want to program for multiple data types in *exactly the same way*. For instance, you may want to define a `square` function that works for both `int`s and `double`s. The operation is identical – `square(x)` should return `x * x` – but any given function must take and return either `int`s or `double`s. To allow reuse of the same code, C++ provides *templates*, which allow you to define a function generically to work with a broad spectrum of data types. The C++

compiler, a very clever beast, automatically infers from each function call what data type is being passed in, and internally generates a new compiled version of the function for each argument type.

The syntax for a template function declaration is as follows:

```
template< class T >      // or template< typename T >
T square(T number) { return number * number; }
```

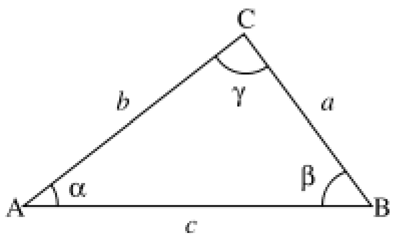
`T` is called a *formal type parameter* – a generic, unspecified type representing whatever type the function happens to be called with. In the copy of the function generated from the function call `square(10)`, all references to `T` would be replaced by `int`. In the call `square(10.0)`, `T` would translate to `double`.

4. Use templates to generalize your array-based `maximum` function to allow anything the compiler can treat as a number – `double`, `int`, `char`, etc. Give sample function calls for two different data types, and indicate what the value of your formal type parameter would be for each.

Part 4 – Toying with Triangles:

Out parameters:
C++ does not allow returning multiple values. One common method of circumventing this language limitation is to pass in reference arguments. Instead of setting a variable equal to the return value of the function, the caller declares several variables and passes them by reference into the function. By changing the values of these variables (called *out parameters*), the callee can simulate returning multiple values.

1.
    a. Give a function prototype for a function that takes as arguments the lengths of two sides of a triangle, the angle in between them in radians, and three out parameters. The out parameters should be set up to allow the function to simulate returning three values: the length of the remaining side and the size of the two remaining angles in radians.
    b. Now write the body of the function. You will probably want to make use of the law of cosines – $c^2 = a^2 + b^2 - 2ab\cos(\gamma)$, with a, b, c, and γ as defined in the diagram above. You may also want to use the `cos`, `acos`, `pow`, and `sqrt` functions from the standard `cmath` include file. It will probably help to first write out the math for each one of the out parameters on a piece of paper.



Recursion:
One common design pattern in creating functions is to set up a function to call itself – to make it a *recursive function*. One classic example is calculating the *n*th term of the Fibonacci sequence: the *n*th term is defined as the sum of term $n-1$ and term $n-2$, except for the first two terms, which are both defined as 1. We can thus define the function `fibonacci` as follows:

```
int fibonacci(int n) {
      if(n == 0 || n == 1)
```

```
            return 1;
        else
            return fibonacci(n - 1) + fibonacci(n - 2);
}
```

`n == 0 || n == 1` is called the *base case* – the situation in which `fibonacci` does not call itself (i.e. it does not *recurse*). Every recursive function must have a base case, and every time the function calls itself the new calls must be one step closer to reaching the base case; otherwise it will keep calling itself forever! (Or at least until the program dies from an overdose of function calls.)

Recursion is a difficult technique to master, and we will not be covering it in-depth for this course. However, it is still worth plowing through one example of defining a recursive function.

2.
   a. One convenient way to define $n!$ is $n(n-1)!$. (The factorial of 0 is 1.) Using this mathematical definition, define a recursive function `factorial` that takes one integer argument and returns an integer. You should first make sure to define a base case – the situation in which you do not need to call `factorial` again to calculate the factorial.
   b. Pascal's triangle is a triangular arrangement of numbers in which each number in the triangle is the sum of the two numbers above it:

```
            1
        1       1
    1       2       1
1       3       3       1
    4       6       4       1
```

and so on. As it happens, the i^{th} element of row n in this triangle can be expressed as $\frac{n!}{(n-i)!i!}$ (or $\begin{pmatrix} n \\ i \end{pmatrix}$, for the mathematically inclined). For the first row, $n = 0$, and similarly $i = 0$ for the first column.

      i. Using your function from part (a), indicate in just a few lines of code how you would print out the i^{th} element of row n. Assume that i and n are defined as integer variables.
      ii. Using the code you wrote for part (i), write a loop that outputs the entire row n.
      iii. Now turn your loop into a function that prints out the nth row of Pascal's triangle. It should take n as an argument, and return nothing.

Part 5 – Arrays:

1. Enter an array of 20 characters and check if each of them is a non-alphanumeric character. Display the character and its position if it is. (You may want to use the `isalnum` function from `<cctype>`).

2. Write a program to input an array of 10 integers and output the average of all the elements of that array.

Passing arrays as arguments:
It is often useful to pass an array as an argument to a function. The syntax for declaring a function that takes such an argument is as follows:

```
int myFunc(int arr[], const int arrayLen, …) { … }
```

The `arr` variable does not need to be declared with a particular length; that is only necessary when *creating* an array. However, the function has no way of knowing how long the array being passed in is, and since all integer arrays are of the same data type, you cannot simply specify that this function takes, say, an array of length 5. The solution is to pass the length of the array as another argument.

Arrays are *always* passed by reference.

As an aside, it is often useful to use named constants to indicate the length of an array – that way, if you change the length of the array, you only need to change the constant in one place. For example:

```
const int NUM_STUDENTS = 47;
double studentGrades[NUM_STUDENTS];
```

3. Adapt your program from Problem 2 to contain a main function that inputs the array, which it then passes to an `average` function. This function should return a decimal value representing the average of the values in the array.

Default initialization of arrays:
You do not need to explicitly initialize all elements of an array of numerical values. If you specify an explicit size for the array, but initialize it with a shorter list, the compiler assumes you want to initialize any unspecified elements to 0. For instance,

```
int numbers[10] = {0};
```

initializes all 10 elements of `numbers` to 0.

4. In the Fibonacci sequence of numbers, each number is the sum of the previous two numbers. Thus, the sequence begins 0, 1, 1, 2, 3, 5, 8, 13, etc. Write a program which creates and displays the first 20 terms of Fibonacci sequence. Use arrays to compute the elements of this series (you may initialize only the first two elements of the array explicitly).

5. Write a program that enters an array of characters from the user and reverses the sequence without creating a new array. (Print the reversed sequence at the end.) For example,
INPUT: a r k a n s a s
OUTPUT: s a s n a k r a

6. Enter an array of characters from the user and remove all the duplicate characters. (You may use a second array for this.) For example,

INPUT: a e s e d f k d m n v s o j n
OUTPUT: a f k m v o j

7. Enter an array of characters from the user and move all the lower case characters to the left side and the upper case characters to the right side of the array. You may want to use the `islower` and/or `isupper` functions from the standard `cctype` include file.

8. Enter an array of 10 integers and sort the first half in an ascending order and the second half in a descending order. For example,
INPUT: 1, 38, 7, 13, 28, 19, 64, 18, 22, 11
OUTPUT: 1, 7, 13, 28, 38, 64, 22, 19, 18, 11